2010 25th Annual IEEE Conference on Computational Complexity

A log-space algorithm for reachability in planar acyclic digraphs with few sources

Derrick Stolee
Department of Computer Science
Department of Mathematics
University of Nebraska–Lincoln
Lincoln, NE, USA
dstolee@cse.unl.edu

Chris Bourke
Department of Computer Science
University of Nebraska–Lincoln
Lincoln, NE, USA
cbourke@cse.unl.edu

N. V. Vinodchandran
Department of Computer Science
University of Nebraska–Lincoln
Lincoln, NE, USA
vinod@cse.unl.edu

***Abstract*—Designing algorithms that use logarithmic space for graph reachability problems is fundamental to complexity theory. It is well known that for general directed graphs this problem is equivalent to the NL vs L problem. This paper focuses on the reachability problem over planar graphs where the complexity is unknown. Showing that the planar reachability problem is NL-complete would show that nondeterministic log-space computations can be made unambiguous. On the other hand, very little is known about classes of planar graphs that admit log-space algorithms. We present a new 'source-based' structural decomposition method for planar DAGs. Based on this decomposition, we show that reachability for planar DAGs with m sources can be decided deterministically in $O(m+\log n)$ space. This leads to a log-space algorithm for reachability in planar DAGs with $O(\log n)$ sources. Our result drastically improves the class of planar graphs for which we know how to decide reachability in deterministic log-space. Specifically, the class extends from planar DAGs with at most two sources to at most $O(\log n)$ sources.**

***Keywords*-reachability; planar graph; acyclic digraph; logspace algorithm;**

I. Introduction

Graph reachability problems are central to complexity theory. The reachability problem of deciding whether there exists a path from a node u to a node v in a directed graph is complete for nondeterministic log-space (NL). In a breakthrough result, Reingold showed that the reachability problem over undirected graphs is complete for deterministic log-space (L) [1]. Various versions of the reachability problem characterize various complexity classes within NL [1]–[4]. Because of its central role in complexity theory, designing algorithms that use logarithmic space for graph reachability problems is a fundamental question.

Planarity has proven to be a very important restriction when dealing with graph problems, both theoretically and algorithmically. Algorithmically, because of certain fundamental structural theorems such as the Lipton-Tarjan planar separator theorem [5], many computational problems over planar graphs admit algorithms with better running time or parallelism. However, from a space-complexity viewpoint progress has started to emerge only recently [6]–[8]. In particular, the space complexity of the reachability problem over planar graphs currently is far from being completely settled. It is known to be hard (under projection reductions) for deterministic log-space [2], but not known to be complete for NL. Recently, it was shown that this problem can be solved in unambiguous logarithmic space (the class UL) [9]. Hence if reachability for planar graphs is complete for NL then all of nondeterministic log-space computations can be made unambiguous (that is NL = UL). While this is very likely, proving NL = UL will be a major result in complexity theory. On the other hand, very little is known about classes of planar graphs that admit log-space algorithms. Jacoby et al. show that various reachability and optimization questions for *series-parallel* graphs admit deterministic log-space algorithms [10], [11]. Series-parallel graphs are a very restricted subclass of planar directed acyclic graphs (DAGs). In particular, such graphs have a single source and a single sink (single source single sink DAGs are sometimes called st-graphs in the literature). Recently, Allender, Barrington, Chakraborty, Datta, and Roy [12] extended Jacoby et al.'s result to show that the reachability problem for planar DAGs with single source and multiple sinks can be decided in log-space. Using a reduction, the authors were able to slightly improve this upper bound to planar DAGs with *two* sources and multiple sinks. This remains the current best class of planar DAGs that admit deterministic logarithmic space algorithms. The present paper makes progress in this direction.

As our main result, we design a deterministic algorithm for reachability in planar DAGs that takes $O(m + \log n)$ space, where m is the number of sources in the input graph. Thus, if the number of sources in the input graph is $O(\log n)$, we get a deterministic log-space algorithm for reachability in planar DAGs.

Theorem 1 (Main Theorem)**.** *The reachability problem for planar directed acyclic graphs with $m = m(n)$ sources is decidable in $O(m + \log n)$ space deterministically.*

Corollary 2. *The reachability problem for planar directed acyclic graphs with $O(\log n)$ sources is in* L.

Our technique begins in Section II by building on Allender et. al.'s technique of decomposing the graph, forming a

1093-0159/10 $26.00 © 2010 IEEE
DOI 10.1109/CCC.2010.36

IEEE computer society

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.

forest in the case of multiple sources. This leads to new classifications of edges using the topological properties of the embedding, defined in Section III. With these topological properties, a game called the Coin Crawl Game is described in Section IV as a high-level description of the reachability algorithm. It starts as a non-deterministic log-space algorithm that can be made deterministic using space linear in the number of sources. The remaining sections describe the algorithm and prove its correctness.

Until specified otherwise, let G be a directed graph on n vertices. Two vertices u, v are specified and the goal of Reach is to compute reachability from u to v in G. By the previous results mentioned earlier, it can be verified in log-space that u and v are in the same connected component of the underlying undirected graph, G is planar, and G has m sources. We delay the recognition of G having no directed cycles until Section VII. Until then, assume that G is acyclic.

Some special notation for edges is given for ease of the arguments. A directed edge $e = xy$ has x as its *tail* and y as its *head*. Superscripts will be used to distinguish the endpoints of e. If the superscript is a number, it will be $e^1 = x$ or $e^2 = y$. If the superscript is a letter, the letter corresponds to the endpoint closest to the object that letter represents. For instance, if an edge e spans two connected components A and B, then e^A is the endpoint in the A component and e^B is the endpoint in the B component. The meaning should be obvious from context.

II. Forest Decomposition

Since G is acyclic, any reverse walk along incoming edges ends at a source. An arbitrary choice of a single incoming edge at each vertex forms a forest F with every tree rooted at the sources. These edges are called *tree edges*. We may assume u is a source, since incoming edges cannot contribute to a $u-v$ path in G. Also, do not select an incoming edge for v and leave it isolated in F. Hence, F has $m+2$ connected components, each a tree rooted at $u, v, s_1, \ldots, s_m$. These trees are called *source trees*, denoted T_x, where x is the root. In figures, each source tree will be shown as a circle with the root labeled in the center. This leads to the visualization that the trees are placed in the plane with the tree edges directed radially away from the source with the leaves on the edge of the circle.

There are some structural relations among vertices in this forest that can be computed in log-space. The source tree containing any vertex a can be found by following the chosen incoming edges until a source is reached. Denote this procedure $s(a)$, where a is in $T_{s(a)}$. The vertices along this walk are called *ancestors* of a.

If two vertices a, b are in the same source tree, their *least common ancestor* can be found. Define $\mathsf{LCA}(a, b)$ to be the vertex x in $T_{s(a)}$ so that x is an ancestor of both a and b and maximizes the number of tree edges between x and $s(a)$.

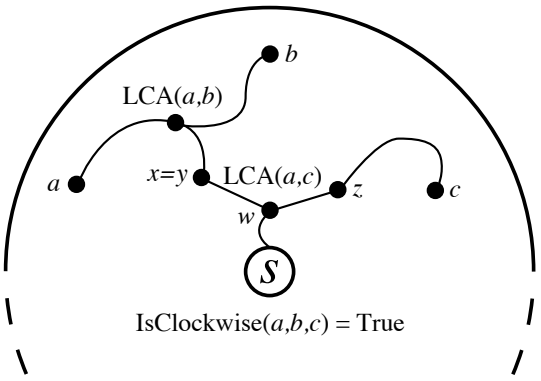


Figure 1. Example of $\mathsf{IsClockwise}(a, b, c)$.

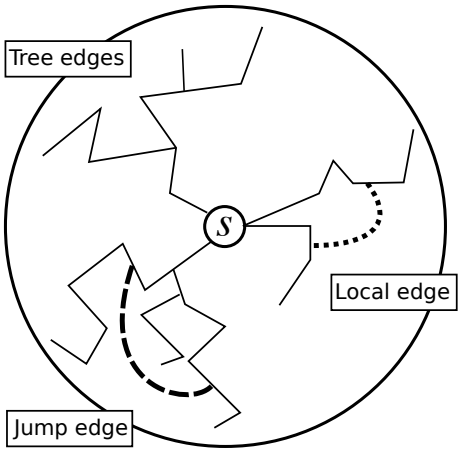
Using the combinatorial embedding of a source tree T_s, a cyclic ordering of the vertices can be computed. For three vertices a, b, c in T_s, let the boolean function $\mathsf{IsClockwise}(a, b, c)$ determine if the triplet a, b, c obeys this cyclic ordering in the clockwise direction, as shown in Figure 1. Let $w = \mathsf{LCA}(a, c)$. If w is not an ancestor of b, then immediately $\mathsf{IsClockwise}(a, b, c)$ is False. Otherwise, let x, y, z be the vertices in the ancestor paths of a, b, c respectively so that $(w, x), (w, y), (w, z)$ are all tree edges. Note that $x \neq z$ by the definition of $\mathsf{LCA}(a, c)$. Now, if x, y, z appear in a clockwise order in the combinatorial embedding of w, then $\mathsf{IsClockwise}(a, b, c) = \mathsf{True}$. This includes the case when $y = x$ or $y = z$.

Any two vertices a, b in the same source tree T_s define two important substructures. First is the *tree path* from a to b, the unique undirected path given by following tree edges from a to $\mathsf{LCA}(a, b)$ and then to b. Second, if ab is an edge not chosen to be in the tree, then a *tree cycle* is defined by combining the tree path from a to b with ab. This tree cycle partitions the plane into two regions, and the number of vertices and sources in each region is countable in log-space. If the count for one of these regions is zero, then it is said that this tree cycle partitions those vertices *trivially*.

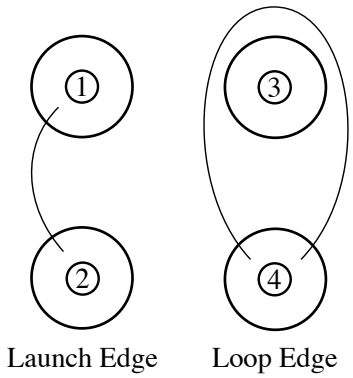
This leads to a method for describing the edges of G. The following edge types partition $E(G)$:

- *Tree edges* are the chosen incoming edges used to define the forest F.
- *Launch edges* are edges between different source trees. The remaining edge types require both endpoints in the same source tree.
- *Local edges* are edges so that the tree cycle partitions the vertices trivially.
- *Jump edges* are edges so that the tree cycle partitions the vertices non-trivially, but partitions the vertices $u, v, s_1, \ldots, s_m$ trivially.
- *Loop edges* are edges so that the tree cycle partitions the sources non-trivially.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.



(a) Tree, Local, and Jump edges.



(b) Launch and Loop edges.

Figure 2. The five edge types in a planar DAG in the forest decomposition.

Examples of these edge types are shown in Figure 2. A path that uses only tree and local edges is called a *local path*. A path that also uses jump edges is called a *jump path*.

The edge types tree, local, and jump are identical to the edge types defined for a Single-source Multiple-sink Planar DAG (SMPD) as in Allender et. al. [12]. They showed that reachability using jump paths was decidable in log-space. Given a vertex x in a tree T_s, two vertices we denote $\mathsf{TreeLeft}(x)$ and $\mathsf{TreeRight}(x)$ can be found in log-space. These represent how far jump paths from x can travel in the counter-clockwise and clockwise directions, respectively. For completeness, we briefly review this algorithm and prove an important property of these vertices as Lemma 3.

By considering different subsets of edge types, we can solve reachability in steps. First, reachability using local paths can be decided by using the subgraph formed by tree and local edges and noting that all sinks are on the same face. Adding a sink that all of these sinks can reach does not change x, y reachability, but forms a Single-source Single-sink Planar DAG (SSPD), which has reachability in L [12]. Given x, vertices $\mathsf{LocalRight}(x)$ and $\mathsf{LocalLeft}(x)$ can be found as the most clockwise and counter-clockwise vertices reachable from x via local paths. These give the initialization of vertices r, ℓ, defining an *explored region* given by each vertex z so that $\mathsf{IsClockwise}(\ell, z, r)$ holds. Now, jump edges may have tail in this region, but head outside. In this case, take the jump edge yz where z is closest to the explored region of all jump edges. Expand the explored region by setting $r = \mathsf{LocalRight}(z)$ or $\ell = \mathsf{LocalLeft}(z)$, depending on which direction yz leaves the explored region. When this process stabilizes and has no jump edges leaving the explored region, define $\mathsf{TreeRight}(x) = r$ and $\mathsf{TreeLeft}(x) = \ell$.

Lemma 3. *Consider r, ℓ during any iteration of the SMPD algorithm. If an edge $e = yz$ has $\mathsf{IsClockwise}(\ell, y, r) = \mathsf{True}$ and $\mathsf{IsClockwise}(\ell, z, r) = \mathsf{False}$, then y is reachable from x.*

Proof: The hypothesis supposed that y is in the explored region, but z is not. Since G is planar, e cannot cross the tree paths from the source to r or ℓ. There also exist paths from x to r and ℓ that e cannot cross. Hence, y is not in the closed curve given by these paths (or else z would be in the explored region). This implies that y is a descendant of the path from x to r or the path from x to ℓ, and thus is reachable from x. When r and ℓ are updated using such a jump edge, the new values are reachable from x using jump paths. ■

Denote by H the *contracted graph of G*: the multigraph formed by contracting each source tree T_s into the root vertex s. This graph may have multiedges and loops. The planar embedding of G induces a planar embedding of H. In the next section, the edges of H are classified by the induced planar embedding. This is a key insight to controlling the launch and loop edges in the reachability algorithm.

III. Topological Equivalence

Consider two edges e_1, e_2 with common endpoints in the contracted graph H. These edges form a simple closed curve in the planar embedding and hence partition the vertices of H into two disjoint subsets (ignoring the endpoints). This partition is *trivial* if one set is empty. If the partition is trivial, we say e_1 and e_2 are *topologically equivalent*. This defines an equivalence relation among the edges of H. Let $[e]$ denote the equivalence class represented by the edge e. Examples of such equivalent pairs are shown in Figure 3.

The contracted graph H can be reduced using these equivalences. First, loops in H form their own simple closed curves. If a loop forms a curve that partitions the vertices trivially, this loop is *trivial* and is removed from H. Remove all edges except a single representative from each class. This requirement of a single representative of each equivalence class renews the definition of the contracted graph H.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.

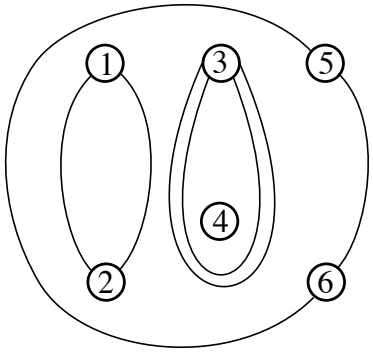


Figure 3. Equivalent edge pairs in a planar multigraph.

The concept of topological equivalence holds for any planar multigraph. The following theorem shows that the maximum number of equivalence classes for an n-vertex multigraph is the same as the maximum number of edges in an n-vertex simple graph.

Theorem 4. *Let $X = (V, E)$ be a planar multigraph. If there are n vertices, there are at most $3n - 6$ topological equivalence classes of edges.*

Proof: We can assume that X has a single representative of each topological equivalence class by removing edges if necessary. We can also assume that X has a maximal number of equivalence classes among all planar supergraphs of X, by adding edges whenever possible. Form a graph Y by subdividing each edge of X. Y is a simple planar graph, so Euler's formula holds [13]. That is, for n_Y the number of vertices, e_Y the number of edges, and f_Y the number of faces in the simple planar graph Y, $n_Y - e_Y + f_Y = 2$. Since $n_Y = n_X + e_X$, $e_Y = 2e_X$ and $f_Y = f_X$, we have

$$2 = (n_X + e_X) - (2e_X) + (f_X) = n_X - e_X + f_X.$$

Each face in X has at least three edges on its border. The only way it can have fewer is if a two multiedges with the same endpoints bounded this face. But, if they are not topologically equivalent, this face contains a vertex. This contradicts maximality, since edges could be placed from this interior vertex to the endpoints of the multiedges.

Since each edge is incident to two faces and each face is incident to at least three edges, $2e_X \geq 3f_X$, which gives $n_X = e_X - f_X + 2 \geq \frac{1}{3}e_X + 2$. Therefore, $e_X \leq 3n_X - 6$. ∎

These topological equivalence classes are represented by launch and loop edges in the graph G. Equivalence can be decided in log-space. For the rest of the paper, consider loop edges to be of launch type as they are treated the same in the algorithms. Let e_1, e_2 be launch edges with endpoints in the trees T_a and T_b. A simple closed curve is formed by combining these edges with the tree path from e_1^a to e_2^a in T_a and the tree path from e_2^b to e_1^b in T_b. Such a curve is shown in Figure 4. The edges e_1, e_2 are topologically equivalent if and only if this curve partitions the sources $u, v, s_1, \ldots, s_m$ trivially.

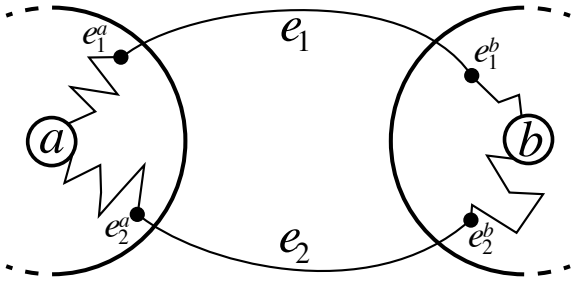


Figure 4. The closed curve for e_1, e_2.

IV. Coin-Crawl Game

In order to provide insight to the algorithm described in Section V, the overall strategy can be condensed into the solution of a game called the *Coin Crawl* game. Consider the contracted graph H and its planar embedding from the previous section. This will act as the board for the game. The vertices of H are represented by disjoint unit circles in the plane, and edges incident to a vertex are distributed evenly around the corresponding circle. The game piece is a coin that can be placed on these unit circles. The coin has an arrow on its face, which will always point to an edge incident to the current vertex. This edge is called the current edge. The game starts with the coin on position T_u. The goal is to reach v through a set of possible moves.

The player's first move rotates the coin so the arrow points at an edge leaving T_u, call it the *starting edge*, e_s. Then, the coin moves to the tree T_i across e_s and the arrow points again to e_s.

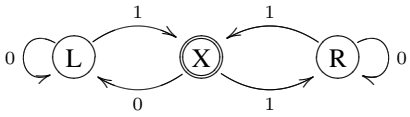
Now, the player can select to rotate the coin Left or Right or select to Cross the current edge. The rotations turn the coin until it points to the next edge in the counter-clockwise (clockwise, respectively) direction from the current edge. The Cross move places the coin on the vertex at the opposite end of the current edge and points the arrow back at this edge. An oracle provides confirmation that these moves are legal, depending on its knowledge of connectivity in G.

A non-deterministic player guesses each move non-deterministically. The player admits failure if the oracle prevents a move, but the player achieves success if the coin reaches v. If there is a successful series of moves, this player will find it.

However, a few promises allow a deterministic player to achieve success as well. First, it is promised that reversing the direction of the coin will not be necessary. That is, if the coin was rotated Right, then a Left move is useless. Moreover, if the coin just crossed an edge, crossing again will not help. This leads to a state machine M describing the possible move sequences as given in Figure 5. Note that

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.

M inputs a binary string $\mathbf{x}$ and outputs a string $M(\mathbf{x})$ of moves from the three defined choices.



$\mathrm{L} = \mathsf{Left} \quad \mathrm{X} = \mathsf{Cross} \quad \mathrm{R} = \mathsf{Right}$

Figure 5. A description of moves.

The second promise is that the portions of the unit circle that the arrow traverses during a rotation do not need to be visited more than once. That is, a player could mark the portions of the unit circles that the arrow crosses and consider the marked portions to be *forbidden zones*. These forbidden zones are shown in Figure 8b on page 7. The length of a move sequence that never revisits the forbidden zones is bounded at some length ℓ. Now, the player can do a brute-force attempt on all starting edges and all move sequences of length ℓ.

The algorithm for reachability with m sources is defined in Section V to follow the non-deterministic player. A data structure called an *explored region* in G is used to represent the coin and its placement in H. An added step called $\mathsf{ExploreClass}$ as defined in Section V-A expands the explored region between moves. The oracle is simulated by the deterministic log-space algorithm $\mathsf{NextClass}$ which determines if there is a path in G that allows the selected move and produces the next edge class. Each move modifies the explored region as described in Section V-B to reflect the rotation or movement of the coin.

To take this non-deterministic method into a deterministic algorithm, Section VI uses the idea of forbidden zones to bound the length of a legal move sequence to at most $12m$. This gives a linear bound on the space required to store such a move sequence, so the deterministic algorithm can be shown to take $O(m + \log n)$ space as given in Theorem 1. The immediate corollary is given that this algorithm runs in log-space for $m = O(\log n)$ sources.

V. Non-deterministic Search

The non-deterministic algorithm relies upon a log-space data structure called the *explored region*. It uses two vertex pointers $A_{\mathrm{L}}, A_{\mathrm{R}}$ to bound the region on the current source tree (side *A*), an edge pointer e_{c} to represent the current class of launch edges, and two vertex pointers $B_{\mathrm{L}}, B_{\mathrm{R}}$ to bound the region on the opposite end of this edge (side *B*). Here, L and R correspond to *left* and *right*, from the conditions that $\mathsf{IsClockwise}(A_{\mathrm{L}}, e_{\mathrm{c}}^{A}, A_{\mathrm{R}})$ and $\mathsf{IsClockwise}(B_{\mathrm{L}}, e_{\mathrm{c}}^{B}, B_{\mathrm{R}})$ will always evaluate as true. A vertex x is said to be *contained in* the explored region if either $\mathsf{IsClockwise}(A_{\mathrm{L}}, x, A_{\mathrm{R}})$ or $\mathsf{IsClockwise}(B_{\mathrm{L}}, x, B_{\mathrm{R}})$ is true. Property 5 states some properties that are required of an explored region.

Property 5. An explored region $C = (A_{\mathrm{L}}, A_{\mathrm{R}}, e_{\mathrm{c}}, B_{\mathrm{L}}, B_{\mathrm{R}})$ must satisfy these conditions:

1) At least one endpoint of e_{c} is reachable from u. Call this vertex z.
2) $A_{\mathrm{L}}, A_{\mathrm{R}}$ are not Null and $\mathsf{IsClockwise}(A_{\mathrm{L}}, e_{\mathrm{c}}^{\mathrm{A}}, A_{\mathrm{R}}) = \mathsf{True}$.
3) If $B_{\mathrm{L}}, B_{\mathrm{R}}$ are not Null, then $\mathsf{IsClockwise}(B_{\mathrm{L}}, e_{\mathrm{c}}^{\mathrm{B}}, B_{\mathrm{R}}) = \mathsf{True}$.
4) If a variable $A_{\mathrm{L}}, A_{\mathrm{R}}, B_{\mathrm{L}}, B_{\mathrm{R}}$ is not Null, then the corresponding vertex is reachable from z using jump paths and edges from $[e_{\mathrm{c}}]$.
5) Any launch edge xy with x contained in C has y reachable from z.

As the algorithm is defined, these properties are shown to persist as the explored region changes as well as used to prove the correctness of the algorithm. Occasionally, C_{A} and C_{B} will be used to refer to the A side or B side of the region.

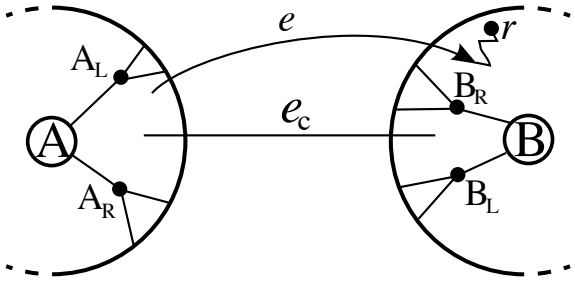
To initialize the explored region C, consider the starting edge e_{s} to be any launch edge leaving T_u. The vertex e_{s}^1 is in T_u and hence reachable. Set $A_{\mathrm{L}} = \mathsf{TreeLeft}(e_{\mathrm{s}}^2)$, $A_{\mathrm{R}} = \mathsf{TreeRight}(e_{\mathrm{s}}^2)$, $e_{\mathrm{c}} = e_{\mathrm{s}}$, and let B_{L} and B_{R} be Null. Property 5 can be seen to hold since reachability from u holds for every vertex in the explored region. Initialize the state machine M to the Cross state.

The algorithm proceeds by non-deterministically selecting a bit x and using the move $M(x)$. The subroutine $\mathsf{NextClass}$ determines if the current explored region allows this move and returns the next edge e_{n}. If the move is not allowed, e_{n} will be Null and the algorithm rejects. Otherwise, C is modified (as described in Section V-B) to reflect the move to e_{n}. Then, the explored region expands by searching through all edges in the topological equivalence class of e_{n} and within C (see subroutine $\mathsf{ExploreClass}$ in Section V-A). Accept when C contains a launch edge to v. Since Property 5.4 holds, v is reachable from u. After the subroutines are defined in the following sections, completeness of this non-deterministic search will be shown by converting any uv path in G to a list of moves that lead this algorithm to v.

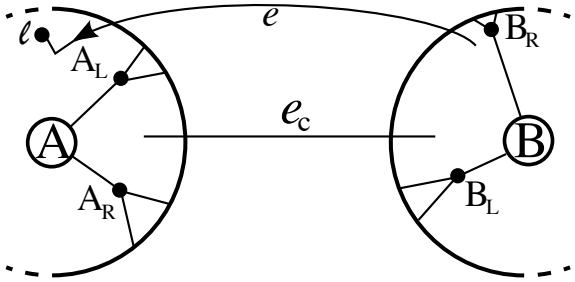
A. Expanding the explored region

The subroutine $\mathsf{ExploreClass}$ expands the explored region C by using jump paths along with launch edges of class $[e_{\mathrm{c}}]$. The launch edges e of class $[e_{\mathrm{c}}]$ with e^1 in C_{A} can be used to expand C_{B} and vice-versa using an alternating iteration procedure. The procedure updates the variables $A_{\mathrm{L}}, A_{\mathrm{R}}, B_{\mathrm{L}}$, and B_{R} and terminates when no change occurs during the update.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.



(a) An A–B iteration.



(b) A B–A iteration.

Figure 6. The alternating ExploreClass iteration types.

First, start with an A–B iteration. Consider all edges $e \in [e_c]$ with e^1 in C_A. Let $\ell = \mathsf{TreeLeft}(e^2)$ and $r = \mathsf{TreeRight}(e^2)$. If B_L and B_R are Null, immediately set $B_L = \ell$ and $B_R = r$. Otherwise if $\mathsf{IsClockwise}(\ell, B_L, e_c^2)$ update B_L to ℓ, as ℓ is farther counterclockwise from e_c than B_L. Similarly if $\mathsf{IsClockwise}(e_c^2, B_R, r)$, then update B_R to r.

The B–A iteration performs the symmetric update for all edges $e \in [e_c]$ with e^1 in C_B. See Figure 6 for an example of these iterations.

Lemma 6. *The subroutine* ExploreClass *preserves Property 5.*

Proof: As long as Property 5.4 holds in a given iteration, the updates to A_L, A_R, B_L, and B_R will respect the first four parts of Property 5, since the edges used for the expansion have reachable endpoints.

To see why Property 5.4 holds after an update, consider a launch edge e with e^1 in C_A. The argument is symmetric when e^1 is in C_B. Without loss of generality, assume $\mathsf{IsClockwise}(A_L, e^1, e_c^A)$, as the argument is symmetric when $\mathsf{IsClockwise}(e_c^A, e^1, A_R)$. By iteration, we can assume that e^1 was not in the explored region until the latest update. Let f_1 be the launch edge that was used to update A_L before the latest update, and f_2 is the launch edge used in the current update.

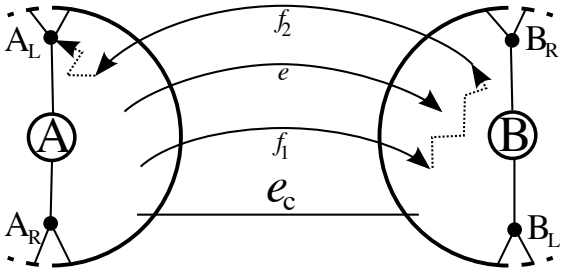
There are two cases to consider.

The first case is that $\mathsf{IsClockwise}(f_1^A, e^1, f_2^A)$. Since e is a launch edge, but appears clockwise between to launch edges of type $[e_c]$, e has class $[e_c]$. If e^1 is reachable from z, e^2 is reachable, so assume that e^1 is not. This means that A_L is not reachable using a path that crosses the tree path from e^1 to $s(e^1)$. Hence, the path from z to A_L must follow a jump path between f_1^B and f_2^B. Figure 7a shows that even though e^1 is not reachable, e^2 is the descendant of this jump path that connects e_c to A_L.

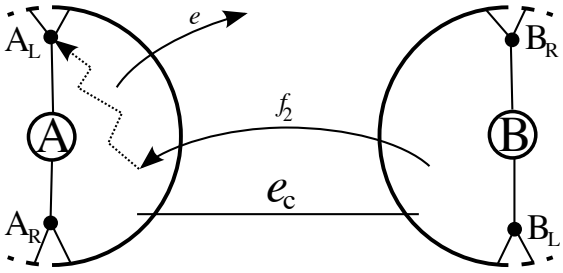
The second case is that $\mathsf{IsClockwise}(e^1, f_1^A, f_2^A)$. That is, e^1 is on the portion of the explored region that is between A_L and f_1^A. However, this region was defined with $A_L = \mathsf{TreeLeft}(f_1^A)$, so by Lemma 3 any launch edge between A_L and f_1^A is reachable from f_1^A and hence from z. This is seen in Figure 7b where the jump path from f_1^A to A_L must cross the tree path from e^1 to $s(e^1)$. Note that any launch edge of a different class than $[e_c]$ will fall into this second case, and hence has both endpoints reachable. ■

B. Moving the explored region

Given a move, it can be determined if that move is legal or not by examining the explored region C.



(a) Edge between updates.



(b) Edge outside updates.

Figure 7. Launch edges within the explored region and how they are reachable. The jagged dotted lines correspond to jump paths between a launch edge and A_L or between two launch edges.

Consider the case of a Cross move. Such a move is legal only if there is a launch edge $e_n \in [e_c]$ so that e_n^1 is in C_A. Property 5.4 shows that e_n^2 is reachable. Modify C_A as follows by swapping A_L with B_L and A_R with B_R. This effectively moves the coin to the opposite side of e_c. Since e_n has e_n^1 on the A side, the subroutine ExploreClass would have initialized B_L and B_R. Property 5 is satisfied.

Now, consider a Right rotation. A Right move attempts to rotate the coin clockwise to the equivalence class of edges adjacent to $[e_c]$. If no other equivalence classes exist on the source tree T_A within the explored region, then this

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.

move is impossible. Otherwise, if two launch edges e_1, e_2 have incident vertices x_1, x_2 in T_{A} and $\mathsf{IsClockwise}(e_{\mathrm{c}}^{\mathrm{A}}, x, y)$ holds, then e_1 is closer to $[e_{\mathrm{c}}]$ than e_2. The edge e that is closest to $[e_{\mathrm{c}}]$ is a representative of the new class and e_{n} is set to e. Moreover, since e_{n} was selected to be closest even among its own equivalence class, there exist edges in $[e_{\mathrm{n}}]$ within C_{A} if and only if e_{n} has an incident vertex within C_{A}. If not, the move is not allowed. Otherwise, set $e_{\mathrm{c}} = e_{\mathrm{n}}$, $z = e_{\mathrm{n}}^{\mathrm{A}}$, and set B_{L} and B_{R} to Null. The modifications for a Left rotation are similar, except use the counter-clockwise direction instead. Since A_{L} and A_{R} are unchanged, B_{L} and B_{R} are Null, and $e_{\mathrm{n}}^{\mathrm{A}}$ is reachable, Property 5 is verified.

VI. Bounding the depth of an accepting path

Let $P = u, x_1, x_2, \ldots, x_k, v$ be a directed path in G. P is called *irreducible* if for every $i, j \in \{1, \ldots, k\}$ with $i < j$, if x_i is an ancestor of x_j in the forest F, then $x_i, x_{i+1}, \ldots, x_{j-1}, x_j$ are the vertices in the tree path from x_i to x_j. That is, if any vertex of P has an ancestor that appears earlier in P, then P follows the tree path. Any path P that is not irreducible can produce an irreducible path by removing the subpath between violating pairs of vertices and replacing it with the corresponding tree path.

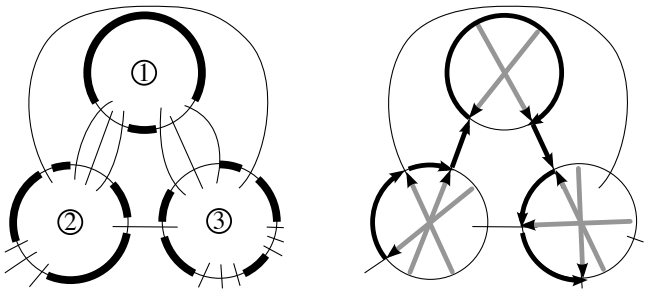
This irreducible property provides the concept of forbidden zones in the Coin Crawl game. Let $P = u, x_1, x_2, \ldots, x_k, v$ as before. If $i_1, \ldots, i_\ell$ are the indices so that (x_{i_j}, x_{i_j+1}) is a launch edge for every j, then the subpaths $x_{i_j+1}, x_{i_j+2}, \ldots, x_{i_{j+1}}$ are jump paths. If (x_{i_j}, x_{i_j+1}) and $(x_{i_{j+1}}, x_{i_{j+1}+1})$ are launch edges of different equivalence classes, all vertices between x_{i_j+1} and $x_{i_{j+1}}$ in that source tree are either ancestors or descendants of the jump path between them. Hence, if P is an irreducible path, then P cannot visit this portion of the source tree without violating the irreducible condition (by visiting a descendant) or the acyclic condition (by visiting an ancestor).

Lemma 7. *An irreducible u–v path induces a list of moves of length at most $12m$.*

Proof: The equivalence classes of launch edges partitions the source trees into two types of regions:

1) Vertices between the tree paths from equivalent launch edges to their sources.
2) Vertices between the tree paths from non-equivalent launch edges to their sources.

These regions are demonstrated in Figure 8a. Each rotation move corresponds to P traversing an entire region of the second type, marking a forbidden zone in the Coin Crawl Game as seen in Figure 8b. Each class of launch edges has two such regions on each side of the class. Since one is used on the rotation move that allows the coin to reach this class, and another is used on the rotation move that allows the coin to leave this class. Hence, this class can be visited at most twice by the move string from an irreducible path. This bounds the number of rotation moves to be at most twice the number of equivalence classes. By Theorem 4, there are at most $3(m+2) - 6 = 3m$ classes and hence at most $6m$ rotation moves.



(a) Partitions of source trees. (b) Forbidden zones.

Figure 8. Partitions of source trees compared to forbidden zones in the Coin Crawl Game.

Since each Cross move must immediately follow a rotation move, the number of Cross moves is also bounded by $6m$. Therefore, the total number of moves is at most $12m$. ∎

Lemma 7 provides the last step to showing Theorem 1. This is shown by combining the non-determinstic log-space algorithm with the $12m$ space required for a move string. Setting $m = O(\log n)$ gives Corollary 2, that reachability for log-source planar DAGs is in L.

VII. Finding Cycles

It is important to recognize that a planar digraph is acyclic, in order to verify that the algorithm executes correctly. Note that by performing the forest decomposition and applying the SMPD algorithm, one can verify in log-space that the tree edges do not form a cycle and each induced source tree is acyclic. It remains only to verify that no cycles exist that use launch or loop edges. If a cycle exists using a launch edge xy, there exists an irreducible path P from y to x. This path can be found using the given algorithm. By iterating over all launch edges and testing for reverse reachability, an existing cycle will be found in log-space.

VIII. Future Work

While this work has increased the class of graphs that allow deciding reachability within deterministic log-space, there are directions that can be pursued for further advancement. The current reachability algorithm has lower bounds on the length of move strings if the graph and forest decomposition are chosen arbitrarily. However, if the forest decomposition was chosen in a non-trivial way it may be possible to reduce the number of moves to sublinear in m. This will translate to an increased number of sources allowed while staying in log-space.

The forest decomposition technique may be useful in other problems involving directed acyclic graphs. More importantly, the topological equivalence of edges in the

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.

resulting contracted graph may have applications to other problems.

ACKNOWLEDGEMENTS

The authors would like to thank the UNL Discrete Math Seminar participants, especially Stephen G. Hartke, for meaningful discussion that helped to correct mistakes and clarify non-obvious proofs. Also, thanks to the anonymous referees whose comments improved this paper. This work was supported by the NSF grants CCF-0430991 and CCF-0830730.

REFERENCES

[1] O. Reingold, "Undirected connectivity in log-space," *J. ACM*, vol. 55, no. 4, 2008.

[2] K. Etessami, "Counting quantifiers, successor relations, and logarithmic space," *Journal of Computer and System Sciences*, vol. 54, no. 3, pp. 400–411, June 1997.

[3] D. A. M. Barrington, C.-J. Lu, P. B. Miltersen, and S. Skyum, "Searching constant width mazes captures the AC^0 hierarchy," in *15th International Symposium on Theoretical Aspects of Computer Science (STACS)*, ser. Volume 1373 in Lecture Notes in Computer Science. Springer, 1998, pp. 74–83.

[4] D. A. M. Barrington, "Bounded-width polynomial-size branching programs recognize exactly those languages in NC^1," *Journal of Computer and System Sciences*, vol. 38, pp. 150–164, 1989.

[5] R. J. Lipton and R. E. Tarjan, "A separator theorem for planar graphs," *SIAM Journal on Applied Mathematics*, vol. 36, no. 2, pp. 177–189, 1979. [Online]. Available: http://www.jstor.org/stable/2100927

[6] E. Allender and M. Mahajan, "The complexity of planarity testing," *Information and Computation*, vol. 189, pp. 117–134, 2004.

[7] S. Datta, R. Kulkarni, N. Limaye, and M. Mahajan, "Planarity, determinants, permanents, and (unique) perfect matchings," in *Proceedings of 2nd International Computer Science Symposium in Russia CSR*, 2007, pp. 115–126, springer-Verlag Lecture Notes in Computer Science series Volume 4649.

[8] S. Datta, N. Limaye, P. Nimbhorkar, T. Thierauf, and F. Wagner, "Planar graph isomorphism is in log-space," *Annual IEEE Conference on Computational Complexity*, vol. 0, pp. 203–214, 2009.

[9] C. Bourke, R. Tewari, and N. V. Vinodchandran, "Directed planar reachability is in unambiguous log-space," *ACM Trans. Comput. Theory*, vol. 1, no. 1, pp. 1–17, 2009.

[10] A. Jakoby, M. Liśkiewicz, and R. Reischuk, "Space efficient algorithms for directed series-parallel graphs," *J. Algorithms*, vol. 60, no. 2, pp. 85–114, 2006.

[11] A. Jakoby and T. Tantau, "Logspace algorithms for computing shortest and longest paths in series-parallel graphs," in *FSTTCS*, 2007, pp. 216–227.

[12] E. Allender, D. A. M. Barrington, T. Chakraborty, S. Datta, and S. Roy, "Planar and grid graph reachability problems," *Theor. Comp. Sys.*, vol. 45, no. 4, pp. 675–723, 2009.

[13] D. B. West, *Introduction to Graph Theory*, 2nd ed. Prentice-Hall, 2001.

Authorized licensed use limited to: University of Nebraska - Lincoln. Downloaded on April 16,2026 at 21:18:10 UTC from IEEE Xplore. Restrictions apply.